ASSIGNMENT 3: INTERACTIVE VISUALIZATION

Ivan Aristy — iae225 *CS-GY 6313 B*

1 Interactive Visualization

Note: For the puposes of page limits, I am treating code blocks as figures; hence, they should not be considered.

1.1 Question

How has my champion's win rate changed over time? Is my champion still strong in the current meta?

I want the user to easily see the win rate of their champion over time. Winrate is a key metric in League of Legends, as it is a good indicator of how strong a champion is in the current meta. This would allow the user to see how their champion has been impacted in recent patches.

Winrate is not a direct representation of power, since some champions are harder to play than others. Easy champions usually have high winrates, while hard champions have lower winrates. Additionally, hyper-specific champions have really high winrates, since they are only played in specific situations.

Nevertheless, patterns in winrates can be indicative of power. For example, a 2% growth in winrates over a patch can be indicative of a very strong relating buff, while a 2% decrease can be indicative of a strong relative nerf. Additionally, a winrate that is consistently above 50% can show that a champion has and is strong in the current meta.

1.2 Data

1.2.1 Data Source

Riot's Developer API is very hard to use. In my opinion, it is not well documented, and it is very hard to get the data you want. Hence, I wll scrape some data off the internet to get the winrate of a champion over time.

Particularly I need:

1. The winrate of a champions over time.
2. The tier of the champion over time.
3. Visual Assets of the champion.

1.2.2 Data Retreival

To get winrate data, I will scrape `https://lolalytics.com/lol/tierlist/` for the winrate and tier of a champion over time.

After analyzing the beautiful soup output, the relevant information for a champion is contained in a div with the class `flex justify-around border border-[#333333] p-2 text-center`. For the next div, we will skip the explanation, but it uses a similar process to the code below.

```
  container_div = soup.find('div', class_='flex␣justify-around
␣␣border␣border-[#333333]␣p-2␣text-center')
  if container_div:
      # Find all the individual sections within this container
      sections = container_div.find_all('div', recursive=False)

      for section in sections:
          value_div = section.find('div', class_='mb-1␣font-bold')
          if value_div:
              value = value_div.get_text(strip=True)
```

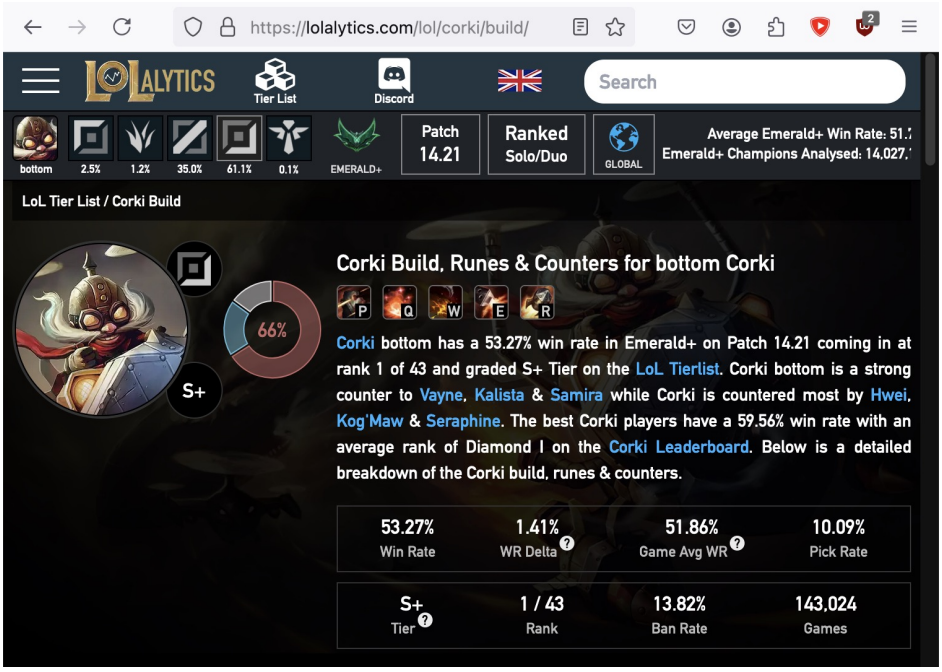

Figure 1: Corki's champion page.

```
                row_1_data.append(value)
print(row_1_data)
```

In Figure 1 the relevant data would be the eight values from "Win Rate" to "Games". However, this information is only for one patch, so the backend function will also receive a parameter to indicate the patch to scrape.

1.2.3 Serving Backend Server

We are using FastAPI to expose the data to the frontend. To accurately model the data, I created the following interface:

```
class ChampionInstance(BaseModel):
  name: str
  patch: float
  win_rate: float
  win_rate_delta: float
  modified_winrate: float
  pick_rate: float
  tier: str
  rank: int
  ban_rate: float
  games: int
```

and exposed the following function to the frontend:

```
@app.get("/test/{champion_name}&{patch_version}")
async def test(champion_name: str, patch_version: str):
  testChampion: ChampionInstance =
    await api.get_champion_data(champion_name, patch_version)
  return testChampion
```

All the functions used were made asynchronous, since load times were a complaint I received from my peers while showing them the assignment.

The benefit of using live data is that, whenever a new patch is released, the user can see how their champion has been impacted by the patch. Additionally, for the current patch, changes in data are reflected in real time.

1.3 Visualization

1.3.1 Frontend Setup

The frontend is a simple React application that uses the `D3` library to create the chart, and react hooks to update and keep track of state (dynamic reloading of data depending on user selected parameters).

For the main App component, we define a useEffect hook that will achieve multiple "Quality of Life" improvements for the user:

- Updating the chart whenever the user selects a new champion, metric, or patch range.
- Displaying a loading spinner while the data is being fetched.
- Asynchronous fetching of data to speed up the loading process.
- Invalidate erroneous data but keep plotting the chart.

```
useEffect(() => {
    async function fetchData() {
      setLoading(true);

      const patchRange = patches.slice(
        patches.indexOf(startPatch),
        patches.indexOf(endPatch) + 1
      );

      try {
        const results = await Promise.all(
          patchRange.map(async (patch) => {
            const response = await fetch(
              `http://127.0.0.1:8000/test/${selectedChampion}&${patch}`
            );
            if (!response.ok)
              throw new Error(`Failed to fetch data for patch ${patch}`);

            const result = await response.json();
            return { patch, value: result[selectedMetric] };
          })
        );

        setData(results);
      } catch (error) {
        console.error("An error occurred while fetching data:", error);
      } finally {
        setLoading(false);
      }
    }

    setData(null);
    fetchData();
  }, [selectedChampion, selectedMetric, startPatch, endPatch]);
```

1.3.2 Visualization Logic